UCRL-ID-149835

Design of Wrapper Integration within the DataFoundry Bioinformatics Application

J. Anderson, T. Critchlow

August 20, 2002

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

DISCLAIMER

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

Design of Wrapper Integration within the DataFoundry Bioinformatics Application

J. Anderson, T. Critchlow

20th August 2002

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

Contents

List of Figures

Abstract

The DataFoundry bioinformatics application was designed to enable scientists to directly interact with large datasets, gathered from multiple remote data sources, through a graphical, interactive interface. Gathering information from multiple data sources, integrating that data, and providing an interface to the accumulated data is non-trivial. Advanced techniques are required to develop a solution that adequately completes this task. One possible solution to this problem involves the use of specialized information access programs that are able to access information and transmute that information to a form usable by a single application. These information access programs, called wrappers, were decided to be the most appropriate way to extend the DataFoundry bioinformatics application to support data integration from multiple sources. By adding wrapper support into the DataFoundry application, it is hoped that this system will be able to provide a single access point to bioinformatics data for scientists. We describe some of the computer science concepts, design, and the implementation of adding wrapper support into the DataFoundry bioinformatics application, and then discuss issues of performance.

1 Introduction

This paper aims to discuss the useful details regarding the design and implementation of wrapper technology within the DataFoundry bioinformatics application. In doing so, various technologies, their role in the implementation of wrapper support, performance issues, and the future work that can be done within the DataFoundry application will be discussed. Technologies that found their way into implementing wrapper support include XML, Java, and information wrappers. Our discussion of implementation and design will focus on extensibility, ease of integration, communication pathways, and performance. Before all of these issues are able to be addressed, however, it is necessary to discuss the background of the problem.

2 Background & Motivation

The DataFoundry bioinformatics application was designed to allow scientists to directly interact with large datasets, gathered from multiple remote data sources, through a graphical, interactive interface. This type of application is of obvious necessity, as there are more than 500 sources of publicly available bioinformatics information available today [1]. Having a central access point to these sources of data would provide many benefits to scientists, including higher productivity and

hopefully more accurate data due to information correlation. There are problems, however, that arise when attempting to design and implement a system to harness numerous sources of bioinformatics information; gathering information from multiple, heterogeneous data sources, integrating that data, and providing an interface to the accumulated data is non-trivial. Advanced techniques are required to develop a solution that is able to solve this problem in anything but the most naïve manner.

The need to access even a small fraction of the publicly available bioinformatics information on the World Wide Web motivated the development of a data access solution that would facilitate the utilization of external data within the DataFoundry application. The end goal of this application is to provide a single access point for scientists to access bioinformatics data. To accomplish this goal it would not be acceptable to hard-code hundreds of different access methods (one for each external data source) into the DataFoundry application itself. Doing this would lead to a huge code-base; it would be unmaintainable, and exceedingly cumbersome to extend in the future. It was decided that an appropriate method to solve the problem of accessing hundreds of external data sources was through the use of wrappers, which could then be used by the DataFoundry server to correlate the data from all of the external data sources for which wrappers exist.

3 Technology

The majority of bioinformatics information available on the World Wide Web is in a format that is relatively non-standard (to a computer's eye), and difficult to access. This information is often only obtainable after giving specific input parameters to a query page, waiting (which, surprisingly, can be done in many ways), and finally being transfered to a page with your desired result. All of the steps between starting a query and retrieving the results, and even processing those results, are non-standard [2].

Wrappers help to alleviate some of the difficulty in accessing heterogeneous data sources from an application developer's perspective. Wrappers are software agents that are able to access, and internalize, a specific data source. Because they are able to internalize the data found at a data source, they are also able to externally represent this data in any desired format. The power of wrappers now becomes evident; because wrappers are able to export the information that was gathered from a specific data source in any format, it becomes possible to correlate the data from heterogeneous data sources by simply defining a common export format. Simplistically, this is all that is needed to correlate an unlimited number of heterogeneous sources.

Figure 1: XML for an Automobile

```
<automobile>
  <make>Ford</make>
  <model>Model T</model>
  <year>1908</year>
</automobile>
```

Figure 2: XML for a Car

```
<car>
  <brand>Toyota</brand>
  <model>Camry</model>
  <year>1984</year>
</car>
```

In order to access the hundreds of bioinformatics data sources found on the world wide web, hundreds of wrappers with similar export rules must be developed, one for each source (or set of sources with homologous interfaces). The wrappers created for use within the DataFoundry bioinformatics application were specifically designed so that each one exports its source specific data in XML-formatted text.

XML (the eXtensible Markup Language) is a language comprised of tagged text [3]. While this paper is not intended to cover XML, a simple example of XML-formatted data, and the help XML lends to the problem of data integration is in order. A simple example of XML can be seen in Figure 1. This example illustrates a simple block of XML describing a 1908 Ford Model T automobile. The power XML offers is the ease with which data can be extracted and converted once in XML format. If presented with a second block of XML text (as in Figure 2), it would be relatively simple to develop multiple schemes for integrating, or displaying as one, the two blocks of information. For example, a mapping could be created to associate `<automobile>` with `<car>` and so on, until one could programmatically integrate the two examples into Figure 3.

Performing such a mapping, when done correctly, reorganizes the delimiters of the data, but maintains the integrity of the data represented. After reorganization, a simple application (that

Figure 3: Integrated XML for a Vehicle

```
<car>
  <make>Ford</make>
  <model>Model T</model>
  <year>1908</year>
</car>
<car>
  <make>Toyota</make>
  <model>Camry</model>
  <year>1984</year>
</car>
```

only knows one representation format, not two) could be used to display the sum of the information. The result of mapping our data like this would be data formatted with a common ontology. An ontology is a "specification of a representational vocabulary for a shared domain of discourse", that is, a defined way to describe a given domain [4]. With vehicles, it could be agreed upon (potentially) that one should use 'car' instead of 'automobile'. Thus, in order to get a single format, one would have to map all variations encountered to the desired ontology. Without spending immense amounts of time maintaining and updating ontology mappings, it would be difficult to automatically integrate thousands of pieces of information from hundreds of wrappers. We will come back to this problem when discussing the design and implementation of the DataFoundry client.

To understand the following discussion of the design and implementation of the DataFoundry bioinformatics application wrapper integration, it is useful if the reader understands the proceeding explanation of wrappers and XML. It might prove mildly helpful as well if the reader has a passing familiarity with the Java programming language and is aware of Java applet/servlet technology.

4 Design & Implementation

The DataFoundry bioinformatics application was first designed in order to access a local database in order to retrieve its bioinformatics data. This allowed for the development of a graphical interface for the available bioinformatics data. For example, an interface usable for BLAST information was created. BLAST, which stands for Basic Local Alignment Search Tool, is a bioinformatics tool able to recognize alignment similarity between sequences [5]. There were limitations built into the use of a local database; namely, the DataFoundry application was unable to utilize the hundreds of publicly available sources of bioinformatics data, including numerous BLAST sources. The data available to the application was limited to the capacity and quality of the local database's information.

The DataFoundry bioinformatics application is actually two applications, a client program, and a server program. This separation allows scientists to move from computer to computer, using the software when they like, without the overhead of needing to re-install both the server and client. The client and server are written in the Java programming language; the client application is a Java applet, which has the capability of running within a user's web browser such that a scientist can go nearly anywhere and still access data with relative ease. For the remainder of this discussion, the terms client and applet will be synonymous, as will servlet (a Java server running

in a servlet engine) and server.

This next section will focus on the design and implementation of the wrapper integration within both the applet and servlet that comprise the DataFoundry bioinformatics application. Both designs are different, and each implementation serves a separate function. The client is designed to retrieve and display results of queries that were issued by scientists, whereas the server is designed to answer those queries by harnessing its available wrappers, and to facilitate ease of extensibility. Within the applet there are problems that arise such as result integration and management of the returned results, while within the server there are separate issues, including wrapper management, communication, and performance. These issues, and more, will be discussed in the following sections.

4.1 Server

As mentioned previously, the DataFoundry application is comprised of a server and a client. This section aims to discuss important aspects of the server-side design of the DataFoundry application. It is easiest to discuss the design and implementation of wrapper integration within the DataFoundry server while focusing on the lifetime of a wrapper-based query. For discussion, we will define the lifetime of a wrapper-based query as the time between receiving an incoming query and the end of the server's query handling mechanism (which will soon be discussed).

Because it was desired that the previous generation of code able to access the local database still be available for queries, we were able to create an entirely different handler for wrapper-based queries. Any query requiring the service of a wrapper will now be processed by this handler. There was a great degree of modularity gained by being able to split the query processing along the lines of legacy versus wrapper. The largest benefit is that new wrapper handling code can be developed while maintaining previous functionality in the DataFoundry servlet.

In order to discuss the new wrapper handling, it is best to look at a schematic of how a wrapper-based query is processed. Figure 4 illustrates the anatomy of a wrapper-based query. From there server's perspective, there are now (since wrapper support has been integrated) two types of queries: wrapper-based, and local database queries. In designing the servlet, this aided in gaining modularity in the code for the wrapper support.

As is evident from Figure 4 the first step taken when handling a wrapper-based query is to start a new thread (see Figure 4, step 3) that will be able to control the instantiation of new wrappers and the communication channel to the client applet. Directly following the instantiation of the

Figure 4: Diagram of a wrapper-based query

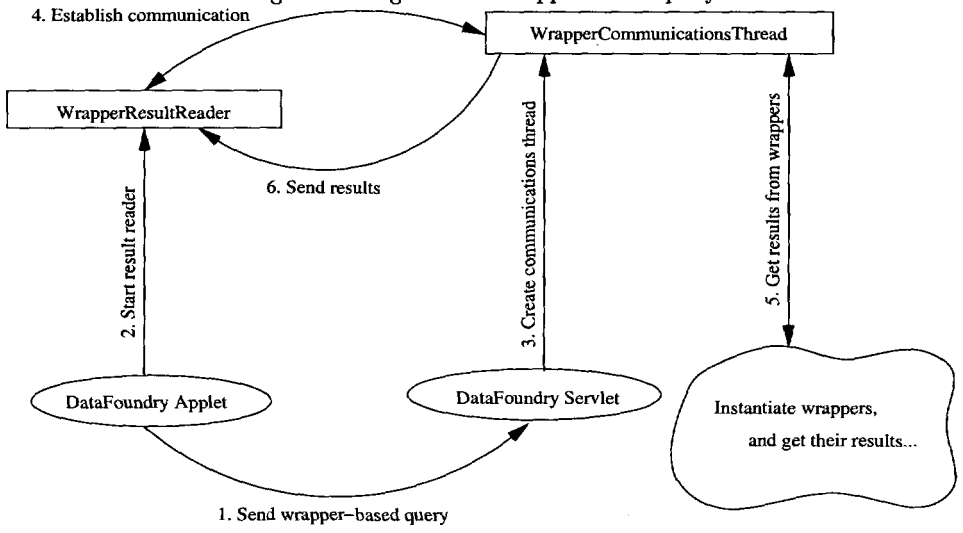


Figure 5: Example XML configuration file for WrapperHandler class

```
<content>
  <wrapper>
    <class>KeywordWrapper001</class>
    <timeout>300</timeout>
    <service>keyword</service>
  </wrapper>
  <mapping>
    <service>keyword</service>
    <handler>KeywordWrapperThread</handler>
  </mapping>
</content>
```

"communications thread", the original server loop continues, ready to service new queries. The communications thread replies to the applet with a message indicating the network port that the applet should connect to in order to retrieve the results of its wrapper-based query. The design, but not the implementation, of the rest of the wrapper-handling on the server-side is straight forward at this point. The server instantiates the appropriate wrappers for a query, and continues looping over the new wrappers until results are found, at which point it sends those results to the applet. This loop continues until all of the wrappers are complete. At this point, the lifetime of a wrapper-based query is finished.

Certain issues remain to be answered, however, regarding how the communications thread knows what wrappers to instantiate. In our design, we allowed for another class (the WrapperHandler) which was able to parse an XML configuration file that defined the wrappers and their service types. A simple configuration file that defines a single wrapper and it's associated handler is shown in Figure 5. The WrapperHandler builds from this file an index of the wrappers available for all the service types it knows about (also from the configuration file). By doing this, the

communications thread is able to query which wrappers should be used to answer a query of a given service type (e.g. keyword or BLAST). Because some of the wrappers that were available at the time of the wrapper integration project did not support the same Application Programming Interface (API), we were forced to develop handler classes for different services of wrappers. These handlers are a layer in between the actual wrapper transformation code, and the communications thread that facilitate the invocation of the wrapper code.

4.2 Client

The design of the client was also driven by the communication path of a wrapper-based query, however, there were also separate issues, such as result presentation, that arose. The first necessity, of course, was that the applet had to be able to retrieve results from the server. The original design was to have the server connect to the client; the client could open a listening port, tell the server to connect, and the wait until results were available. While this approach worked to a degree, there was a major limitation: firewalls. Because of the direction of the connection, our first approach only worked when both the client and the server were not blocked by a firewall. In order to solve this problem, we simply reversed the order of connection (the client connects to the server), and set the connection to keep alive until the server disconnects. The connection model that we decided upon is very similar to that of passive File Transfer Protocol (FTP) connections [6].

The portion of the applet that connects to the servlet is actually the WrapperResultsReader (see Figure 4, step 2). This thread connects to the servlet as described above, and continues to read in result objects until the socket through which it is communicating has been closed by the servlet. At this point, the result reader invokes a callback function with the results as its argument.

The useful portions of the applet, from a scientists perspective, are now ready to be discussed. Depending on the type of the query, whether searching keywords in bioinformatics databases or using BLAST to search for sequence homology, different callback functions are used to decide what interface to build for the newly returned results. To date, there are two interfaces for bioinformatics data within the DataFoundry application: BLAST and keyword. The BLAST interface, as mentioned above, was created when the local database was the only bioinformatics source accessible to the applet. As proof that wrappers could be seamlessly integrated, the BLAST interface was not changed as a result of adding wrapper support to the application. Because of this, it is not a very interesting topic for discussion relating to wrapper integration. The keyword interface, however, is an interface that depends heavily on wrappers providing the 'keyword'

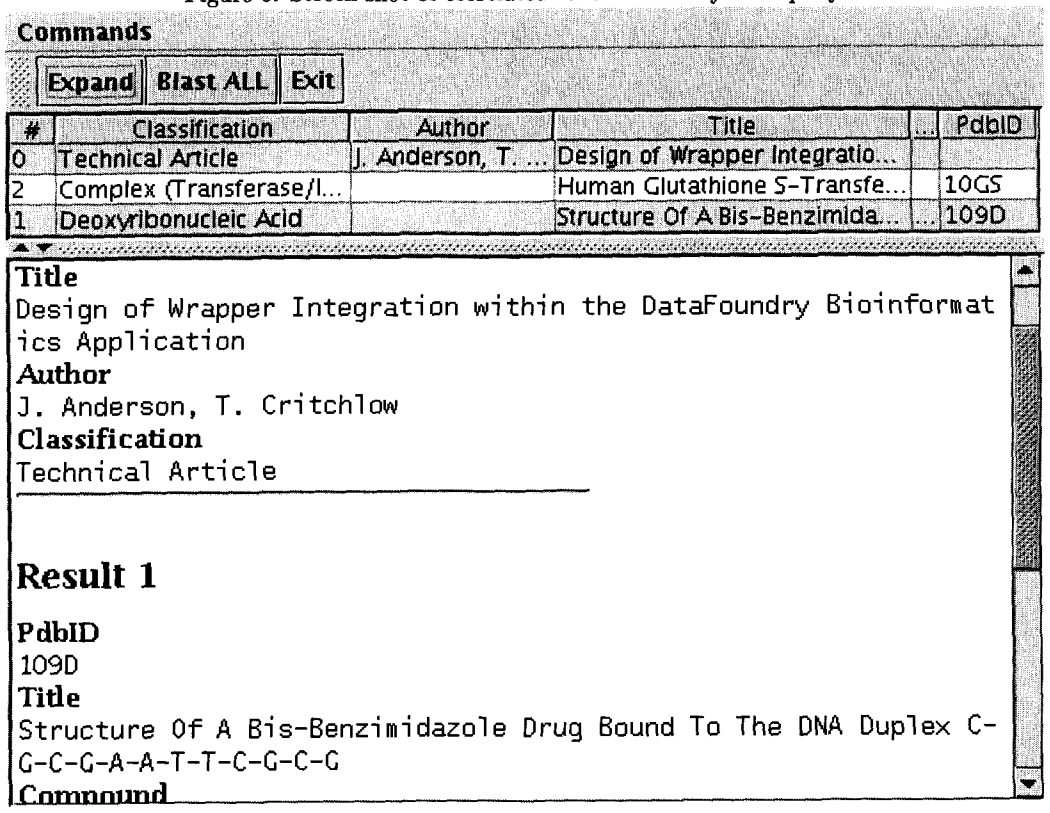
service.

Keyword wrappers are able to search external bioinformatics data sources using an interface similar to a search engine. These wrappers, however, do not necessarily return information that is similar in nature; one source might search books on bioinformatics, while another might search protein databases. Remembering the discussion earlier regarding ontologies, the fact that we might have XML fields named 'Sequence' and 'DNASequence' in the same result set presented a problem. We had to develop a user interface that was robust enough to present information in a useful manner.

We decided not to create a common ontology for all of the results in the DataFoundry application itself because it would be as difficult to generate and maintain the mappings describing how to transform the field names in the XML as it is to generate wrappers for external data sources. Instead, the keyword query results interface does not depend on XML field names to match. Ignoring the differences in field names tends to work, because the end goal of wrappers is that they will be automatically generated, and because perfect integration of the keyword results is not needed to create a very usable interface. The advantage of automatically generated wrappers is that a common ontology, in a limited sense, can be generated in the wrappers themselves. When an external source uses the term 'book name', or 'name', a wrapper can use 'title'. By performing this mapping, it is likely that congruent information will correlate, leaving only fields that are unique to a subset of keyword wrappers (e.g. 'ISBN', 'author', etc.) uncorrelated.

The keyword interface uses the above assumptions to build its user interface, an example of which can be seen in Figure 6. This interface is spit into two parts, the summary view, and the expanded view. In the summary view, all of the results from the keyword wrapper query are displayed in a table. This table has one row per result, and the columns are comprised of qualifying fields from the XML of that result. Because it is possible that fields within a result could be very long, it is necessary to apply some sort of heuristic to limit the length of field displayed in the summary table. At this time, if the average length of the field over all of the results is below a defined threshold, then that field is displayed in the summary view. The fact that some fields might not be displayed by default motivates the expanded view. When a user decides to expand a result, that result is displayed, with all of its fields, in the expanded view pane.

Figure 6: Screen shot of correlated results from keyword query



5 Performance & Discussion

While the design of the wrapper integration as discussed above proves to be very flexible, there are certain aspects of the system that create performance issues within the system. The current system works very well when there are a limited number of wrappers used per query issued to the server. However, when hundreds of wrappers are instantiated per query, and multiple scientists are running queries, the scalability of the server designed as it is comes into question. Imagine a situation when 100 scientists are using the same server to perform their queries, each issues 10 queries, each query against 100 wrappers. The first problem with this situation lies in the multi-threaded nature of running wrappers. When a wrapper is run, a new Java thread is started. This works well when there are a limited number of wrappers being started, however instantiating hundreds (or thousands) or Java threads concurrently tends to choke even the best servlet engine. A second problem involved with running hundreds of concurrent wrappers is that of network bandwidth; the servlet has to access all of the external data sources prior to running the transformations that the wrapper dictates. In addition, the wrapped information must then be sent to the client application. This problem is linear in nature; running more wrappers will increase the network bandwidth needed to obtain acceptable performance.

There are also performance problems that the client faces when running a query against a large number of wrapped sources. These problems, not surprisingly, map almost directly to those faced on the server-side. The results returned from the server are in XML format, and because of this the first step taken by the applet is to parse the XML data. The method used in the DataFoundry application to parse the XML is a DOM tree. This eases the implementation of new query types (e.g. keyword and BLAST), however it takes a large amount of memory. If the applet's virtual machine is unable to allocate enough memory to parse and store a DOM tree for the returned data, the applet will not be able to function properly. Another problem with the current design of the applet is its naïve method of displaying a large number of results. Currently, support for incremental result viewing is not supported, and because if this it is imaginable that with enough wrappers it would be possible to obtain overwhelming amounts of data. This poses problems to both the network (retrieving too much data at once), and the user interface (displaying too much of the data for it to be usable).

In the future most of these problems will likely be fixed. A few methods that could be employed to address the problems described above will now be discussed. The problem of creating too many threads on the server-side could be reduced by defining a maximum number of concurrently

running wrapper threads (perhaps dependent on the server machine's capabilities). After this limit is reached, the remaining wrappers would be started when running threads finish until all wrappers are finished. This solution would limit the number of threads running at one time within the servlet. There are issues involved with implementing this control. Depending on the maximum thread number (M), and the number of wrappers for the query (W), it is possible that when W is greater than M, the total time taken to complete the query will increase. Wrappers W-M through W will have to wait until currently running wrappers finish before starting, thus increasing the total time versus running all of the wrappers directly with no control. On the positive note, the server is more likely to continue operating under heavy usage patterns after a control such as above is implemented.

The issues of network, memory, and display could likely be solved with an intelligent implementation of result segmentation. Because the results of each wrapper are returned from the server to the applet once they are available, it makes sense to implement segmented, or incremental, result browsing in the client. By doing so one would negate some of the performance penalty incurred by implementing the above thread regulation, as well as ease the memory used during the creation of the XML DOM tree on the client side (it is likely that fewer DOM trees will be in memory concurrently). Also, by incrementally displaying results, one would not have to worry about presenting too many results at one time to the scientist.

6 Conclusion

The current design and implementation of wrapper support within the DataFoundry bioinformatics application succeeds in many ways. We were able to reuse the graphical interface already in place for BLAST results, as well as easily implement a new user interface for keyword queries that worked seamlessly with the newly added wrapper support. The current design also allows for different uses of wrappers in the future; it is so general that non-bioinformatics interfaces could easily be written, given non-bioinformatics wrappers. In the area of extensibility and flexibility we have also succeeded. Because the wrapper communication path between the client and server is abstracted away from the use of the data and the methods in which the data is retrieved, it is easy to extend and improve the communication path. In the future it should be simple to address issues of performance by making changes to the communication pathway between the client and server, leaving most of the rest of the system untouched.

References

[1] DBCAT, The Public Catalog of Databases. *http://www.infobiogen.fr/services/dbcat/*, August 2002.

[2] D. Buttler, and T. Critchlow. Using Meta-Data to Automatically Wrap Bioinformatics Sources. Objects, XML, and Databases OOPSLA Workshop. 2001.

[3] Extensible markup language (XML) 1.0 Technical Report (Second Edition), W3C, 2000.

[4] T. R. Gruber. A Translation Approach to Portable Ontology Specifications. 1993.

[5] WU BLAST Archives. *http://blast.wustl.edu/*, August 2002.

[6] J. Postel, and J. Reynolds. "File Transfer Protocol (FTP)", RFC 959, ISI, October 1985.